The Puma Operating System For Massively Parallel Computers *

Lance Shuler,
Chu Jong, Rolf Riesen, David van Dresser
- Sandia National Laboratories

Arthur B. Maccabe, Lee Ann Fisk
- The University of New Mexico

T. Mack Stallcup
- Intel SSD

Abstract

This paper describes the Puma operating system, the successor to the Sandia and University of New Mexico operating system (SUNMOS). Puma is a multiprocessor operating system that is made up of two basic structures, the quintessential kernel (Q-kernel) and the process control thread (PCT). Together, they provide the user with a flexible, lightweight, high performance, message passing environment for massively parallel computers. In this paper, we discuss the structure of Puma and its unique message passing design based on portals.

Keywords: SUNMOS, Puma, operating systems, message passing

1 Introduction

This paper describes the Puma message passing operating system that is under development at Sandia National Labs and the University of New Mexico. Puma is the successor to the Sandia and University of New Mexico operating system (SUNMOS). Puma inherited many of SUNMOS's goals including small size, high performance, and scalability. Puma is a multiprocessor operating system that has expanded message passing capability through a new message passing mechanism known as portals. A portal is essentially an opening in the user's address space, from which or to which another node may read or write data.

This paper describes the structure of Puma and the portal message passing concept. Section 2 covers the structure of Puma. Section 3 describes the trust mechanism of Puma. Section 4 covers the message passing mechanism using portals and Section 5 provides a summary.

2 Puma Structure

The Puma operating system can be broken down into two operating system entities known as the quintessential kernel (Q-kernel) and the process control thread (PCT). Together, they provide the user with the support needed for a high performance, multiprocessor, message passing environment.

2.1 Rationale

Figure 1 shows the structure of the Puma operating system. The criteria that motivated this form of the Puma design are as follows:

- The Q-kernel must occupy minimal space. (Small)
- The Q-kernel must remain alive through software faults. (Persistent)
- The Q-kernel must not grow in size as the number of nodes increases. (Scalable)

*This work was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-76DP00789

Shrinking the size of the kernel necessitates the moving of some kernel level tasks and structures out of kernel space and into user space. Specifically, most structures relating to message passing were moved into the user process' space. In addition, the functionality covering the management of system resources was moved out of the kernel and put under the responsibility of the process control thread or PCT. The message passing side of Puma will be discussed later in more detail. The two sections below will discuss the relationship between the Q-kernel and the PCT.

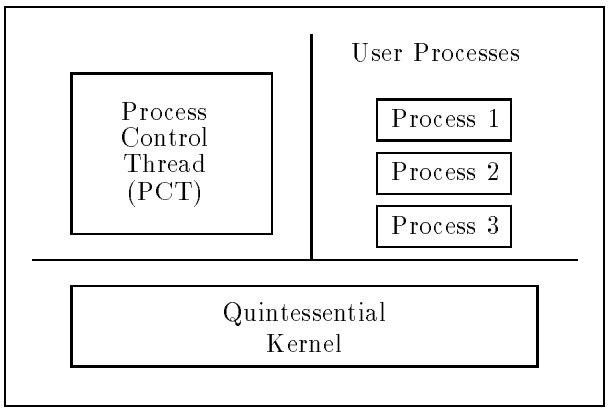

Figure 1: Puma Structure

2.2 Quintessential Kernel

Figure 1 shows the Q-kernel as the lowest level of the operating system. It sits on top of the hardware and performs hardware services on behalf of the PCT and user processes.

The Q-kernel supports the minimal set of tasks that require execution in supervisor mode. Some of these tasks include handling interrupts and handling hardware faults. In addition, the Q-kernel performs supervisor mode services requested by the PCT such as message dispatch and reception, context switching, virtual address manipulation, and running a process. The Q-kernel also handles message dispatch and reception on behalf of user processes.

The Q-kernel does not manage system resources, however. For instance, the Q-kernel does not perform process loading, manage process virtual address spaces, or do job scheduling. The Q-kernel's primary function is to handle hardware interrupts and exceptions, and to perform specific tasks on behalf of the PCT and user processes that must be executed in supervisor mode.

By removing the management tasks from the kernel level, we succeed in building a kernel that is small and that has a minimal set of clearly defined tasks to perform. Such a kernel is simple to maintain, fast, and reliable in the sense that because of its small size, it has a much smaller window in which a failure can occur.

2.3 Process Control Thread

Figure 1 shows the PCT in user space sitting on top of the kernel, yet seperate from other user processes. The PCT is essentially a user level thread with special privileges. It has read/write access to all memory in user space and is in charge of managing all operating system resources. This involves process loading, job scheduling, and memory management. In addition, the PCT may initiate contact with an available server on behalf of a user process.

The PCT and the Q-kernel must work together to provide a complete operating system. The PCT will decide what physical memory and virtual addresses a new process is to have and at the behest of the PCT, the Q-kernel will setup the virtual addressing structures for the new process that are required by the hardware. The PCT will decide which process is to run next and at the behest of the PCT, the Q-kernel will flush caches, setup the hardware registers, and run this process. Notice the clear seperation between resource management and kernel task execution. The PCT acts as manager and the Q-kernel acts as hardware request server.

2.4 Features

Some attractive features result by splitting operating system functionality between the Q-kernel and the PCT. These include fault tolerance and multiple resource management policies.

The small size and well defined tasks of the Q-kernel make reliable software fault tolerance realizable. Even if the PCT faults for some reason, the Q-kernel can remain alive ensuring that the node remains alive. For large Paragons, such as the one at Sandia, this means large savings in downtime, since many software faults that take down a node require a reboot of the whole machine as well as the reloading of currently running applications. In the scenario we present here, the PCT simply needs to be reloaded, and the local running applications need to be restarted.

Another attractive feature is the flexibility that can be achieved by a modular PCT. There can be only one PCT per node. However, one may have several PCTs to chose from. One could have a PCT that is single tasking, a PCT that is multitasking, a PCT that

does priority scheduling, a PCT that does round robin scheduling, a development PCT that supports debugging and profiling, and a performance PCT that supports no debugging or profiling. All of these would sit on top of the same Q-kernel, but would have different management policies.

In addition, a site could have two machine partitions, one that runs a performance PCT that is single tasking with no debugging and one that runs a development PCT that is multitasking and supports debugging and profiling. Repartitioning requires simply reloading PCT's in a different configuration, rather than rebooting the whole machine.

3 Trust

Puma is based on a system of multiple levels of trust. The Q-kernel trusts other Q-kernels. A corollary to this is that the Q-kernel trusts the network, since no message will enter the network except by way of another Q-kernel. Q-kernels do not trust PCTs or user processes.

The PCT trusts the Q-kernel and other PCTs. PCTs do not trust user processes. User processes trust both the Q-kernel and the PCT.

High performance is achieved through this trust model. Since the Q-kernel trusts the network, messages do not have to go through a source verification process. They can be immediately deposited in user space. This significantly cuts time out of the latency path.

Address verification is necessary in the Q-kernel, but only at a few well defined set of kernel entry points.

4 Message Passing

Puma message passing is based on a new concept known as a portal. A portal is essentially an opening in an application's address space to which data can be directly read from or directly written to using message passing. Portals are designed to avoid memory copies that are required when there is send side or receive side message buffering. In addition, portals provide a way for the Q-kernel to reply to read requests without having to perform a context switch and move to user mode. Memory copies and upcalls to user mode are costly operations. Portals are designed to avoid both. The sections below describe the portal message passing concept and present the basic portal message passing structures.

4.1 Portals

Portals map messages into user space. Figure 2 illustrates the mapping. A portal is referenced through an index into a portal table, where each table entry refers to either a match list or a memory descriptor. A memory descriptor describes the layout of the memory associated with the portal. Matching lists provide portals with a matching semantic that can be used to bind specific messages to specific memory regions. Each entry of the matching list has a memory descriptor associated with it. A portal may refer to a memory descriptor directly or to multiple memory descriptors indirectly through a matching list. It should be noted that due to issues of scalability, the portal table, matching lists, and memory descriptors all reside in user space, rather than in kernel space.

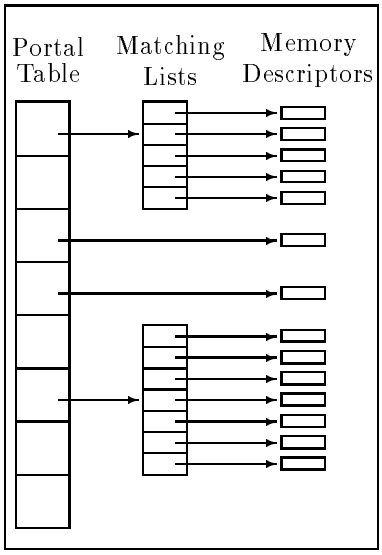

Figure 2: Portal Structure

4.1.1 Memory Descriptors

Memory descriptors describe the layout of the memory associated either with the portal descriptor or with a match list entry. For the discussion in this section, the way the memory descriptor is attached to the portal is irrelevant. Puma supports four types of memory layout.

- Dynamic layout
- Single block layout
- Combined block layout
- Independent block layout

There are also several options associated with memory descriptors that are considered orthogonal to describing memory layout. One option that has already been mentioned is having a matching semantic by using the matching lists. In addition, a memory descriptor may be designated as read only or as write only.

An additional option is being able to specify either sender or receiver managed offsets for read and write operations. When the sender manages the offset, messages can specify an offset into the portal where data should be read from or should be written to. This is particularly useful when implementing multinode servers that work together to service an application's I/O requests. With receiver managed offsets, the receiver increments the offset each time data is read from or written to a memory descriptor. Since data is incrementally read or written from the beginning of the memory block to the end, it is important that read/write requests arrive in the proper order.

There are some memory descriptors that describe the layout of memory in terms of an array of buffers. There is an option for viewing these buffers as a linear list or as a circular list of buffers.

Another option allows the user to choose whether or not to acknowledge (ack) write operations. The ack includes information describing which write operation was applied to the memory descriptor on the receiving end. The possibilities for write operations are:

- Save header
- Save body
- Save header and body

Each memory descriptor may allow only a subset of these orthogonal options. However, matching semantics is the one option common to all memory descriptors. The semantics of the different memory descriptors and their associated options are described below.

Dynamic Figure 3 illustrates the organization of a dynamic memory descriptor. The memory descriptor points to a block of memory that is laid out in the form of a heap. Within the heap, the Q-kernel maintains a linked list of free memory blocks and a linked list of arrived messages. When the Q-kernel receives a message destined for this type of memory descriptor, the it mallocs memory out of this heap, deposits the message, and adds the message onto the end of the arrived messages list. A message may be unlinked and freed by the application at the user level.

Regarding orthogonal options, the dynamic memory descriptor may not be read from, and hence is write only by default. The acknowledgement option on write operations is allowed. Sender and receiver managed offsets are not supported and neither is the save-body write operation. The save-header and save-header-and-body operations are supported.

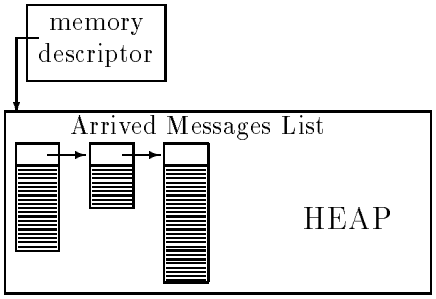

Figure 3: Dynamic Memory Descriptor

Single Block Figure 4 illustrates the organization of a single block memory descriptor. The memory descriptor points to a single contiguous block of memory. This memory descriptor was designed to permit multiple servers to fill in the single block of memory. Since multiple messages may contribute a piece of data, it doesn't make since to save each header, particularly since the receiver doesn't know and doesn't really care how many servers are being used. As a result, no headers are saved in a single block memory descriptor as a matter of policy.

Regarding orthogonal options, the single block memory descriptor may be setup as read only or as write only. This single block structure is designed to take advantage of having multiple servers that manage I/O and other services. Hence, sender or receiver managed offsets are supported with this memory descriptor. The save-header and save-header-and-body write operations are not supported, but the save-body operation is supported. In addition, acknowledging write operations is supported.

Independent Block Figure 5 illustrates the memory structure in an independent block memory descriptor. The memory descriptor points to an array

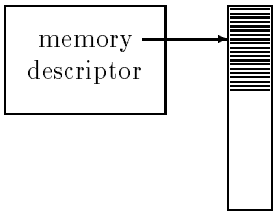

Figure 4: Single Block Memory Desciptor

of buffer descriptors, each of which points to a contiguous block of memory. As the title implies, each block is treated as an independent entity. One and only block of memory is permitted per message. In addition, the list of buffers is traversed in order. If the next available buffer is not large enough to hold the arrived message, then the Q-kernel stops immediately and flags an error bit, even though there may be messages further down the array that are large enough to consume the message.

Supported orthogonal options include setting up the memory descriptor as read only or as write only. In addition, the array of buffers in the memory descriptor may be designated as linear or circular. Offsets are not supported in independent block memory descriptors. Write operations that are supported include save-header and save-header-and-body. Finally, acknowledging write operations is supported.

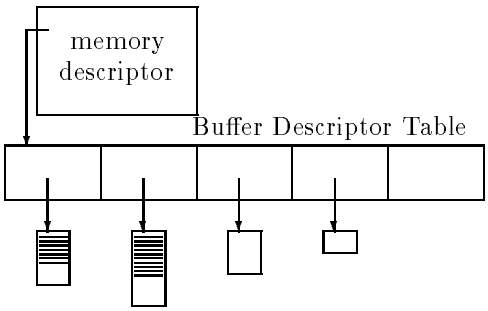

Figure 5: Independent Block Memory Descriptor

Combined Block Figure 6 illustrates the memory structure in a combined block memory descriptor. The memory descriptor points to an array of buffer descriptors, each of which points to a contiguous block of memory. The title implies that the memory blocks are combined in some fashion. Specifically, this memory descriptor is intended to perform exactly the same as the single block memory descriptor, except that instead of containing memory laid out as a contiguous block, the combined block memory descriptor has memory laid out as a *logically* contiguous block. In reality, the logically contiguous block is made up of an array of memory segments that may not be contiguous.

This memory descriptor supports the scatter and gather message passing operations. A single message destined for this memory descriptor that has a length equal to the sum of all the lengths of the blocks will be split up and placed into the blocks according to their order in the array (scatter operation). In the case of the scatter, the memory descriptor would be setup as a write only. On the otherhand, the memory descriptor could have the same block structure for a gather operation, except be read only. A read request on this memory descriptor for a length equal to the sum of all the blocks in the combined block memory descriptor would result in a single message reply consisting of all the memory blocks assembled according to their order in the buffer descriptor array.

Like the single block memory descriptor, multiple servers may read or write to this memory as though it were one contiguous block. No headers are saved for the same reason that they are not saved in single block memory descriptors. In addition, this memory descriptor supports the same orthogonal options as the single block memory descriptor.

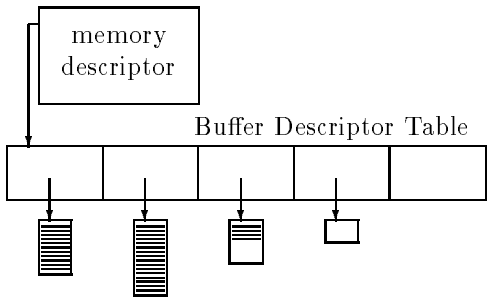

Figure 6: Combined Block Memory Descriptor

4.1.2 Matching List

The matching list makes it possible to have matching semantics for portals at the kernel level. Matching semantics provide a way to prepost receives, so that a user has better control of directing which message is to be deposited where.

When a message arrives at a portal that supports matching, the Q-kernel searches down the match list until an entry is found where the following fields match in both the match list entry and in the message header:

- source group id
- source group rank
- 64 match bits

Depending on the protocol, all of these fields may be wildcarded in the match list entry. On a successful match, the Q-kernel deposits the message into the memory descriptor associated with the matching list entry.

At a particular entry, the Q-kernel could experience one of the following failures:

- Failure on no match
- Failure on no buffer
- Failure on no fit

Failure on no match means that there was a failure to match on one of the above listed match fields. Failure on no buffer means that there is no available buffer in the memory descriptor for the message. Failure on no fit, means that there may be memory in an available buffer of the memory descriptor, but it is not big enough to hold the message.

Handling these failures requires three-way overflow branching out of the match list entry. On each of the failures, the user may specify where to go next. Typically, failure on no match is handled by going on to the next match list entry. Handling failure on no buffer or no fit depends on the protocol. The protocol could designate that the message be dropped by pointing to the end of the list. On the otherhand, the protocol could point to an overflow memory descriptor that simply saves the message header for future use. The matching list provides a great deal of flexibility in handling overflow conditions and is not restricted to the cases mentioned here.

4.2 Portal Issues

We mentioned earlier that all the message passing structures are in user space. This makes it easy for users to manage the portal structures without having to pay the cost of trapping into the Q-kernel. In addition, this allows the Q-kernel to remain small even though the number of processes per node may be large.

There are some issues associated with having message passing structures in user space, however, that must be addressed. At any time, an application may intentionally or accidentally corrupt the portal structures that the Q-kernel may be accessing. Thus, the Q-kernel must have the proper address verification mechanisms available to protect itself against the application. It is a general Puma policy to allow an application to harm itself (including writing over its own stack), but it cannot be allowed to hurt the Q-kernel, PCT, or other running applications.

Another problem with having the user manipulate portal structures rather than the Q-kernel is the problem of race conditions. For instance, suppose a portal is setup with a matching list that points to posted receives and an overflow buffer. The user level receive will first check to see if there is a matching message in the overflow memory discriptor. If there is no such message, then the user level receive will post the message into the matching list. However, during the time that the user level receive is setting up the posted receive, the target message may arrive. The Q-kernel may search through the matching list that still doesn't have the posted receive yet, and place the message in the overflow memory descriptor. Only after the message is tucked away in the overflow, does the post receive operation complete. The message essentially snuck in and the user application never saw it. There are techniques for handling race conditions that will solve such a scenario. Library writers and users who work with portals will need to employ these techniques in order to ensure proper protocol implementations.

5 Summary

In this paper, we described the organization of the Puma operating system. We discussed the structure and cooperative relationship of the Q-kernel and the PCT. We demonstrated how this split operating system structure naturally provides feature benefits in the areas of fault tolerance and flexible resource management policies.

We introduced the fundamental Puma message passing concept known as a portal. We presented the organization of a portal and discussed the various ways of describing memory using portal memory descriptors. In addition, we described the semantics of matching and illustrated how the matching list could be used to handle posted receives and overflow conditions.

We called attention to the advantages and disadvantages of having message passing structures in user space rather than in kernel space. In particular, library writers must be careful to employ basic techniques with portals that will eliminate race conditions. Race conditions may occur anytime a user application and the Q-kernel access the same structures within the same time frame.

Finally, we have presented a comprehensive multiprocessor operating system in Puma that meets the fundamental demands of today's massively parallel community: high performance, scalability, persistence, and light-weight message passing functionality.

This paper and other information is available at the following web site:

http://www.cs.sandia.gov/~ rolf/puma/puma.html

References

[1] Accetta, M.J., et.al. *Mach: A New Kernel Foundation for UNIX Development.* Proceedings of the Summer 19865 USENIX Conference, pp. 93-113, July 1986.

[2] Burns, C.M., Kuhn, R.H., and Werme, E.J., *Low Copy Message Passing on the Alliant CAMPUS/800.* Proceedings of Supercomputing'92, pp. 760-769, November 1992.

[3] Cheriton, D.R., *The V Kernel: A Software Base for Distributed Systems.* IEEE Software, 1(2):19-42, April 1984.

[4] Cheriton, D.R., *The V Distributed System.* Communications of the ACM, March 1988.

[5] Maccabe, Arthur B., and Wheat, Stephen R., *Message passing in PUMA.* Sandia National Laboratories Technical Report SAND93-0935, 1993.

[6] Renesse, R. van, and Tanenbaum, A.S., *Short Overview of Amoeba.* Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures, pp. 1-10, April 1992.

[7] Rozier, M., et.al., *The Chorus Distributed Operating System.* Computing Systems, 1(4), 1988.

[8] Wheat, Stephen R., Maccabe, Arthur B., Riesen, Rolf, van Dresser, David W., and Stallcup, T. Mack. *PUMA: An operating system for massively parallel systems.* Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, pages 56-65. IEEE Computer Society Press, 1994.

[9] Zajcew, R., et.al., *An OSF/1 UNIX for Massively Parallel Multicomputers.* Proceedings of the Winter 1993 USENIX Conference, pp. 449-468, January 1993.